24. Subtyping, Polymorphie und Vererbung

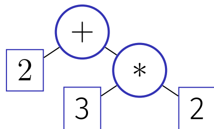
Ausdrückbäume, Aufgabenteilung und Modularisierung, Typhierarchien, virtuelle Funktionen, dynamische Bindung, Code-Wiederverwendung, Konzepte der objektorientierten Programmierung

Letzte Woche: Ausdrucksbäume

- Ziel: Arithmetische Ausdrücke repräsentieren, z.B.

```
2 + 3 * 2
```

- Arithmetische Ausdrücke bilden eine *Baumstruktur*

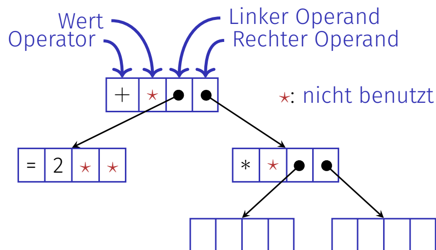


- Ausdrucksbäume bestehen aus *unterschiedlichen* Knoten: Literale (z.B. 2), binäre Operatoren (z.B. $+$), unäre Operatoren (z.B. $\sqrt{\,}$), Funktionsanwendungen (z.B. $\cos$), etc.

Nachteile

Implementiert mittels *eines einzigen* Knotentyps:

```
struct tnode {
  char op; // Operator ('=' for literals)
  double val; // Literal's value
  tnode* left; // Left child (or nullptr)
  tnode* right; // ...
  ...
};
```



Beobachtung: `tnode` ist die „Summe" aller benötigten Knoten (Konstanten, Addition, ...) ⇒ Speicherverschwendung, unelegant

Nachteile

Beobachtung: **`tnode`** ist die „Summe“ aller benötigten Knoten – und jede Funktion muss diese „Summe“ wieder „auseinander nehmen“, z.B.:

```
double eval(const tnode* n) {
  if (n->op == '=') return n->val; // n is a constant
  double l = 0;
  if (n->left) l = eval(n->left); // n is not a unary operator
  double r = eval(n->right);
  switch(n->op) {
    case '+': return l+r; // n is an addition node
    case '*': return l*r; // ...
    ...
```

$\Rightarrow$ Umständlich und somit fehleranfällig

Nachteile

```
struct tnode {
  char op;
  double val;
  tnode* left;
  tnode* right;
  ...
};
```

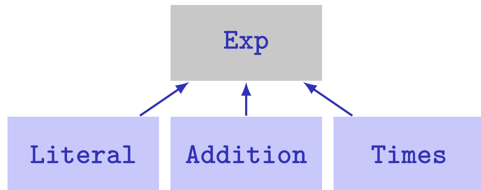
```
double eval(const tnode* n) {
  if (n->op == '=') return n->val;
  double l = 0;
  if (n->left) l = eval(n->left);
  double r = eval(n->right);
  switch(n->op) {
    case '+': return l+r;
    case '*': return l*r;
    ...
```

Dieser Code ist nicht *modular* – das ändern wir heute!

Neue Konzepte heute

1. Subtyping

- Typhierarchie: **`Exp`** repräsentiert allgemeine Ausdrücke, **`Literal`** etc. sind konkrete Ausdrücke
- Jedes **`Literal`** etc. ist auch ein **`Exp`** (Subtyp-Beziehung)



- Deswegen kann ein **`Literal`** etc. überall dort genutzt werden, wo ein **`Exp`** erwartet wird:

```
Exp* e = new Literal(132);
```

Neue Konzepte heute

2. Polymorphie und dynamische Bindung

- Eine Variable vom *statischen* Typ **Exp** kann Ausdrücke mit unterschiedlichen *dynamischen* Typen „beherbergen“:

```
Exp* e = new Literal(2); // e is the literal 2
e = new Addition(e, e); // e is the addition 2 + 2
```

- Ausgeführt werden die Memberfunktionen des *dynamischen* Typs:

```
Exp* e = new Literal(2);
std::cout << e->eval(); // 2

e = new Addition(e, e);
std::cout << e->eval(); // 4
```

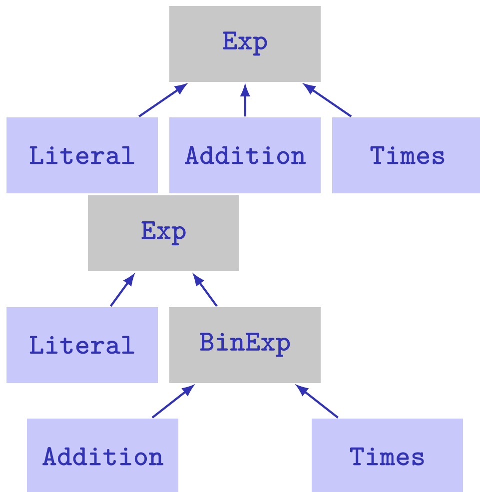
Neue Konzepte heute

3. Vererbung

- Manche Funktionalität ist für mehrere Mitglieder der Typhierarchie gleich
- Z.B. die Berechnung der Grösse (Verschachtelungstiefe) binärer Ausdrücke (**`Addition`**, **`Times`**):

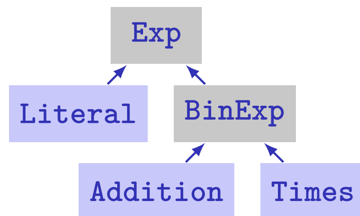
$$1 + size(left\ operand) + size(right\ operand)$$

⇒ Funktionalität einmal implementieren und dann an Subtypen *vererben*



Vorteile

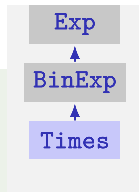
- Subtyping, Polymorphie und dynamische Bindung ermöglichen *Modularisierung durch Spezialisierung*
- Vererbung erlaubt gemeinsamen Code trotz Modularisierung
  $\Rightarrow$ *Codeduplikation vermeiden*




```
Exp* e = new Literal(2);
std::cout << e->eval();

e = new Addition(e, e);
std::cout << e->eval();
```

Syntax und Terminologie



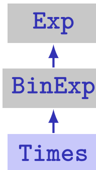

```
struct Exp {
  ...
}

struct BinExp : public Exp {
  ...
}

struct Times : public BinExp {
  ...
}
```

Anmerkung: Wir konzentrieren uns heute auf die neuen Konzepte (Subtyping, ...) und ignorieren den davon unabhängigen Aspekt der Kapselung (`class`, `private` vs. `public` Membervariablen)

Syntax und Terminologie




```
struct Exp {
  ...
}

struct BinExp : public Exp {
  ...
}

struct Times : public BinExp {
  ...
}
```

- **BinExp** ist eine von **Exp** *abgeleitete Klasse*[1]
- **Exp** ist die *Basisklasse*[2] von **BinExp**
- **BinExp** *erbt* von **Exp**
- Die Vererbung von **Exp** zu **BinExp** ist *öffentlich* (**public**), daher ist **BinExp** ein *Subtyp* von **Exp**
- Analog: **Times** und **BinExp**
- Subtyprelation ist transitiv: **Times** ist ebenfalls ein Subtyp von **Exp**

[1] Subklasse, Kindklasse [2] Superklasse, Elternklasse

Abstrakte Klasse `Exp` und konkrete Klasse `Literal`

```
struct Exp {
  virtual int size() const = 0;
  virtual double eval() const = 0;
};
```

…das macht **Exp** zu einer *abstrakten* Klasse

Aktiviert dynamische Bindung

Erzwingt Implementierung durch abgeleitete Klassen …

```
struct Literal : public Exp {
  double val;

  Literal(double v);
  int size() const;
  double eval() const;
};
```

Literal erbt von **Exp** …

…ist aber ansonsten eine ganz normale Klasse

`Literal`: Implementierung

```
Literal::Literal(double v): val(v) {}
```

```
int Literal::size() const {
  return 1;
}
```

```
double Literal::eval() const {
  return this->val;
}
```

Subtyping: Ein Literal ist ein Ausdruck

Ein Zeiger auf einen Subtyp kann überall dort verwendet werden, wo ein Zeiger auf einen Supertyp gefordert ist:

```
Literal* lit = new Literal(5);
Exp* e = lit; // OK: Literal is a subtype of Exp
```

Aber nicht umgekehrt:

```
Exp* e = ...
Literal* lit = e; // ERROR: Exp is not a subtype of Literal
```

Polymorphie: Ein Literal verhält sich wie ein Literal

```
struct Exp {
  ...
  virtual double eval();
};

double Literal::eval() {
  return this->val;
}
```

```
Exp* e = new Literal(3);
std::cout << e->eval(); // 3
```

- *Virtuelle* Memberfunktionen: der *dynamische* Typ (hier: **`Literal`**) bestimmt die auszuführenden Memberfunktionen
  ⇒ *dynamische Bindung*
- Ohne **`virtual`** bestimmt der *statische Typ* (hier: **`Exp`**) die auszuführende Funktion
- Wir vertiefen das nicht weiter

Weitere Ausdrücke: Addition und Times

```
struct Addition : public Exp {
  Exp* left; // left operand
  Exp* right; // right operand
  ...
};
```

```
int Addition::size() const {
  return 1 + left->size()
           + right->size();
}
```

```
struct Times : public Exp {
  Exp* left; // left operand
  Exp* right; // right operand
  ...
};
```

```
int Times::size() const {
  return 1 + left->size()
           + right->size();
}
```

😃 Aufgabenteilung

🤬 Codeduplizierung

Gemeinsamkeiten auslagern ...: `BinExp`

```
struct BinExp : public Exp {
  Exp* left;
  Exp* right;

  BinExp(Exp* l, Exp* r);
  int size() const;
};
```

```
BinExp::BinExp(Exp* l, Exp* r): left(l), right(r) {}
```

```
int BinExp::size() const {
  return 1 + this->left->size() + this->right->size();
}
```

Bemerkung: **`BinExp`** implementiert **`eval`** nicht und ist daher, genau wie **`Exp`**, eine abstrakte Klasse

...Gemeinsamkeiten erben: Addition

```
struct Addition : public BinExp {
  Addition(Exp* l, Exp* r);
  double eval() const;
};
```

Addition erbt Membervariablen (**left**, **right**) und Funktionen (**size**) von **BinExp**

```
Addition::Addition(Exp* l, Exp* r): BinExp(l, r) {}
```

Aufruf des *Superkonstruktors* (Konstruktor von **BinExp**) zwecks Initialisierung der Membervariablen **left** und **right**

```
double Addition::eval() const {
  return
    this->left->eval() +
    this->right->eval();
}
```

...Gemeinsamkeiten erben: `Times`

```
struct Times : public BinExp {
  Times(Exp* l, Exp* r);
  double eval() const;
};
```

```
Times::Times(Exp* l, Exp* r): BinExp(l, r) {}
```

```
double Times::eval() const {
  return
    this->left->eval() *
    this->right->eval();
}
```

Beobachtung: **`Additon::eval()`** und **`Times::eval()`** sind sich sehr ähnlich und könnten ebenfalls zusammengelegt werden. Das dafür notwendige Konzept der *funktionalen Programmierung* geht jedoch über diesen Kurs hinaus.

Weitere Ausdrücke und Operationen

- Weitere Ausdrücke, als von `Exp` abgeleitete Klassen, sind möglich, z.B. $-$, $/$, $\sqrt{\ }$, $\cos$, $\log$
- Eine ehemalige Bonusaufgabe (Teil der heutigen Vorlesungsbeispiele auf Code Expert) veranschaulicht, was alles möglich ist: Variablen, trigonometrische Funktionen, Parsing, Pretty-Printing, numerische Vereinfachungen, symbolische Ableitungen, ...

Mission: Monolithisch → modular ✓

```
struct tnode {
  char op;
  double val;
  tnode* left;
  tnode* right;
  ...
}
```

```
double eval(const tnode* n) {
  if (n->op == '=') return n->val;
  double l = 0;
  if (n->left != 0) l = eval(n->left);
  double r = eval(n->right);
  switch(n->op) {
    case '+': return l + r;
    case '*': return l - r;
    case '-': return l - r;
    case '/': return l / r;
    default:
      // unknown operator
      assert (false);
  }
}
```

```
int size (const tnode* n) const { ... }

...
```

```
struct Literal : public Exp {
  double val;
  ...
  double eval() const {
    return val;
  }
};
```

```
struct Addition : public Exp {
  ...
  double eval() const {
    return left->eval() + right->eval();
  }
};
```

```
struct Times : public Exp {
  ...
  double eval() const {
    return left->eval() * right->eval();
  }
}
```

```
struct Cos : public Exp {
  ...
  double eval() const {
    return std::cos(argument->eval());
  }
}
```

Es gibt noch so viel mehr …

Nicht gezeigt/besprochen:

- Private Vererbung (`class B : ~~public~~ A`)
- Subtyping und Polymorphie ohne Zeiger
- Nicht-virtuelle Memberfunktionen und statische Bindung (`~~virtual~~ double eval()`)
- Überschreiben geerbter Memberfunktionen und Aufrufen der überschriebenen Implementierung
- Mehrfachvererbung (multiple inheritance)
- …

Objektorientierte Programmierung

Im letzten Kursdrittel wurden einige Konzepte der *objektorientierten Programmierung* vorgestellt, die auf den kommenden Folien noch einmal kurz zusammengefasst werden.

Kapselung (Wochen 10-13):

- Verbergen der Implementierungsdetails von Typen (privater Bereich) vor Benutzern
- Definition einer Schnittstelle (öffentlicher Bereich) zum kontrollierten Zugriff auf Werte und Funktionalität
- Ermöglicht das Sicherstellen von Invarianten, sowie den Austausch von Implementierungen ohne Anpassungen von Benutzercode

Objektorientierte Programmierung

Subtyping (Woche 14):

- Typhierarchien mit Super- und Subtypen können angelegt werden um Verwandtschaftbeziehungen sowie Abstraktionen und Spezialisierungen zu modellieren
- Ein Subtyp unterstützen mindestens die Funktionalität, die auch der Supertyp unterstützt – i.d.R. aber mehr, d.h. Subtypen erweitern die Schnittstelle (den öffentlichen Bereich) ihrer Supertypen
- Daher können Subtypen überall dort eingesetzt werden, wo Supertypen verlangt sind …
- …und Funktionen, die auf abstrakteren Typen (Supertypen) operieren können, können auch auf spezialisierteren Typen (Subtypen) operieren
- Die in Woche 7 vorgestellten Streams bilden eine solche Typhierarchie: **`ostream`** ist der abstrakte Supertyp, **`ofstream`** etc. sind spezialisierte Subtypen

Objektorientierte Programmierung

Polymorphie und *dynamische Bindung* (Woche 14):

- Ein Zeiger vom statischen Typ T_1 kann zur Laufzeit auf Objekte vom (dynamischen) Typ T_2 zeigen, falls T_2 ein Subtyp von T_1 ist
- Wird eine virtuelle Memberfunktion von einem solchen Zeiger aus aufgerufen, so entscheidet der dynamische Typ darüber, welche Funktion ausgeführt wird
- D.h.: Trotz gleichem statischen Typ kann beim Zugriff auf eine gemeinsame Schnittstelle (Memberfunktionen) eines solchen Zeigers ein anderes Verhalten auftreten
- Zusammen mit Subtyping ermöglicht es dies, neue konkrete Typen (Streams, Ausdrücke, ...) zu einem bestehenden System hinzuzufügen, ohne dieses abändern zu müssen

Objektorientierte Programmierung

Vererbung (Woche 14):

- Abgeleitete Klassen erben die Funktionalität, d.h. die Implementierungen von Memberfunktionen, ihrer Elternklassen
- Dies ermöglicht es, gemeinsam genutzten Code wiederverwenden zu können und vermeidet so Codeduplikation
- Geerbte Implementierungen können auch überschrieben werden, um zu erreichen, dass eine abgeleitete Klasse sich anders verhält als ihre Elternklasse (im Kurs nicht gezeigt)

25. Zusammenfassung

Zweck und Format

Nennung der wichtigsten Stichwörter zu den Kapiteln. Checkliste: „kann ich mit jedem Begriff etwas anfangen?“

- (M) Motivation: Motivierendes Beispiel zum Kapitel
- (K) Konzepte: Konzepte, die nicht von der Implementation (Sprache) C++abhängen
- (S) Sprachlich (C++): alles was mit der gewählten Sprache zusammenhängt
- (B) Beispiele: genannte Beispiele der Vorlesung

Kapitelüberblick

- 1. Einführung
- 2. Ganze Zahlen
- 3. Wahrheitswerte
- 4. Defensives Programmieren
- 5./6. Kontrollanweisungen
- 7./8. Fliesskommazahlen
- 9./10. Funktionen
- 11. Referenztypen
- 12./13. Vektoren und Strings
- 14./15. Rekursion
- 16. Structs und Overloading
- 17. Klassen
- 18./19. Dynamische Datenstrukturen
- 20. Container, Iteratoren und Algorithmen
- 21. Dynamische Datentypen und Speicherverwaltung
- 22. Subtyping, Polymorphie und Vererbung

1. Einführung

M

- Euklidischer Algorithmus

K

- Algorithmus, Turingmaschine, Programmiersprachen, Kompilation, Syntax und Semantik
- Werte und Effekte, (Fundamental)typen, Literale, Variablen, Bezeichner, Objekte, Ausdrücke, Operatoren, Anweisungen

S

- Include-Direktiven `#include <iostream>`
- Hauptfunktion `int main(){...}`
- Kommentare, Layout `// Kommentar`
- Typen, Variablen, L-Wert `a` , R-Wert `a+b`
- Ausdrucksanweisung `b=b*b;` , Deklarationsanweisung `int a;`, Rückgabeanweisung `return 0;`

2. Ganze Zahlen

Ⓜ
- Celsius to Fahrenheit

Ⓚ
- Assoziativität und Präzedenz, Stelligkeit
- Ausdrucksbäume, Auswertungsreihenfolge
- Arithmetische Operatoren
- Binärzahldarstellung, Hexadezimale Zahlen, Wertebereich
- Zahlendarstellung mit Vorzeichen, Zweierkomplement

Ⓢ
- Arithmetische Operatoren `9 * celsius / 5 + 32`
- Inkrement / Dekrement `expr++`
- Arithmetische Zuweisungen `expr1 += expr2`
- Konversion `int` ↔ `unsigned int`

Ⓑ
- Celsius to Fahrenheit, Ersatzwiderstand

3. Wahrheitswerte

Ⓚ

- Boole’sche Funktionen, Vollständigkeit
- DeMorgan’sche Regeln

Ⓢ

- Der Typ `bool`
- Logische Operationen `a && !b`
- Relationale Operationen `x < y`
- Präzedenzen `7 + x < y && y != 3 * z`
- Kurzschlussauswertung `x != 0 && z / x > y`
- Die **`assert`**-Anweisung, **`#include <cassert>`**

Ⓑ

- Div-Mod Identität.

4. Defensives Programmieren

Ⓚ
- Assertions und Konstanten

Ⓢ
- Die **`assert`**-Anweisung, **`#include <cassert>`**
- **`const int speed_of_light=2999792458`**

Ⓑ
- Assertions für den GGT

5./6. Kontrollanweisungen

Ⓜ

- Linearer Kontrollfluss vs. interessante Programme, Spaghetti-Code

Ⓚ

- Auswahlanweisungen, Iterationsanweisungen
- (Vermeidung von) Endlosschleifen, Halteproblem
- Sichtbarkeits- und Gültigkeitsbereich, Automatische Speicherdauer
- Äquivalenz von Iterationsanweisungen

Ⓢ

- if Anweisungen `if (a % 2 == 0) {..}`
- for Anweisungen `for (unsigned int i = 1; i <= n; ++i) ...`
- while und do-Anweisungen `while (n > 1) {...}`
- Blöcke, Sprunganweisungen `if (a < 0) continue;`
- Switch Anweisung `switch(grade) {case 6: }`

Ⓑ

- Summenberechnung (Gauss), Primzahltest, Collatz-Folge, Fibonacci Zahlen, Taschenrechner, Notenausgabe

7./8. Fliesskommazahlen

Ⓜ
- Richtig Rechnen: Celsius / Fahrenheit

Ⓚ
- Fixkomma- vs. Fliesskommazahldarstellung
- (Löcher im) Wertebereich
- Rechnen mit Fliesskommazahlen, Umrechnung
- Fliesskommazahlensysteme, Normalisierung, IEEE Standard 754
- *Richtlinien für das Rechnen mit Fliesskommazahlen*

Ⓢ
- Typen `float`, `double`
- Fliesskommaliterale `1.23e-7f`

Ⓑ
- Celsius/Fahrenheit, Euler, Harmonische Zahlen

9./10. Funktionen

Ⓜ
- Potenzberechnung

Ⓚ
- Kapselung von Funktionalität
- Funktionen, formale Argumente, Aufrufargumente
- Gültigkeitsbereich, Vorwärts-Deklaration
- Prozedurales Programmieren, Modularierung, Getrennte Übersetzung
- *Stepwise Refinement*

Ⓢ
- Funktionsdeklaration, -definition `double pow(double b, int e){ ... }`
- Funktionsaufruf `pow (2.0, -2)`
- Der typ `void`

Ⓑ
- Potenzberechnung, perfekte Zahlen, Minimum, Kalender

11. Referenztypen

Ⓜ

- Funktion Swap

Ⓚ

- Werte-/ Referenzsemantik, Pass by Value / Pass by Reference, Return by Reference
- Lebensdauer von Objekten / Temporäre Objekte
- Konstanten

Ⓢ

- Referenztyp `int& a`
- Call by Reference und Return by Reference `int& increment (int& i)`
- Const-Richtlinie, Const-Referenzen, Referenzrichtlinie

Ⓑ

- Swap, Inkrement

12./13. Vektoren und Strings

M

- Iteration über Daten: Sieb des Eratosthenes

K

- Vektoren, Speicherlayout, Wahlfreier Zugriff
- (Fehlende) Grenzenprüfung
- Vektoren
- Zeichen: ASCII, UTF8, Texte, Strings

S

- Vektor Typen `std::vector<int> a {4,3,5,2,1};`
- Zeichen und Texte, der Typ char `char c = 'a';`, Konversion nach **`int`**
- Vektoren von Vektoren
- Ströme `std::istream, std::ostream`

B

- Sieb des Eratosthenes, Caesar-Code, Kürzeste Wege

14./15. Rekursion

Ⓜ

- Rekursive math. Funktionen, Das n-Queen Problem, , Lindenmayer-Systeme, Kommandozeilenrechner

Ⓚ

- Rekursion
- Aufrufstapel, Gedächtnis der Rekursion
- Korrektheit, Terminierung,
- Rekursion vs. Iteration
- Backtracking, EBNF, Formale Grammatiken, Parsen

Ⓑ

- Fakultät, GGT, Sudoku-Löser, Taschenrechner

16. Structs und Overloading

Ⓜ

- Datentyp Rationale Zahlen selber bauen

Ⓚ

- Heterogene Datenstruktur
- Funktions- und Operator-Overloading
- Datenkapselung

Ⓢ

- Struct Definition `struct rational {int n; int d;};`
- Mitgliedszugriff `result.n = a.n * b.d + a.d * b.n;`
- Initialisierung und Zuweisung,
- Überladen von Funktionen `pow(2)` vs. `pow(3,3);`, Überladen von Operatoren

Ⓑ

- rationale Zahlen, komplexe Zahlen

17. Klassen

Ⓜ
- Rationale Zahlen mit Kapselung

Ⓚ
- Kapselung, Konstruktion, Mitgliedsfunktionen

Ⓢ
- Klassen `class rational { ... };`
- Zugriffssteuerung `public:` / `private:`
- Mitgliedsfunktionen `int rational::denominator () const`
- Das implizite Argument der Memberfunktionen

Ⓑ
- Endlicher Ring, Komplexe Zahlen

18./19. Dynamische Datenstrukturen

Ⓜ
- Unser eigener Vektor

Ⓚ
- Allokation, Zeiger-Typen, Verkettete Liste, Allokation, Deallokation, Dynamischer Datentyp

Ⓢ
- Die **`new`** Anweisung
- Zeiger `int* x;`, Nullzeiger `nullptr`.
- Adress-, Dereferenzoperator `int *ip = &i; int j = *ip;`
- Zeiger und Const `const int *a;`

Ⓑ
- Verkettete Liste, Stack

20. Container, Iteratoren und Algorithmen

Ⓜ
- Vektoren sind Container

Ⓚ
- Iterieren mit Zeigern
- Container und Iteratoren
- Algorithmen

Ⓢ
- Iteratoren `std::vector<int>::iterator`
- Algorithmen der Standardbibliothek `std::fill (a, a+5, 1);`
- Einen Iterator implementieren
- Iteratoren und const

Ⓑ
- Ausgeben eines Vektors, einer Menge

21. Dynamische Datentypen und Speicherverwaltung

Ⓜ

- Stack
- Ausdrucksbaum

Ⓚ

- Richtlinie „Dynamischer Speicher”
- Gemeinsamer Zeiger-Zugriff
- Dynamischer Datentyp
- Baumstruktur

Ⓢ

- **`new`** und **`delete`**
- Desktruktor **`stack::~stack()`**
- Kopierkonstruktor **`stack::stack(const stack& s)`**
- Zuweisungsoperator **`stack& stack::operator=(const stack& s)`**
- Dreierregel

Ⓑ

- Binärer Suchbaum

22. Subtyping, Polymorphie und Vererbung

Ⓜ

- Erweitern und Verallgmeinern von Ausdrucksbäumen

Ⓚ

- Subtyping
- Polymorphie und dynamische Bindung
- Vererbung

Ⓢ

- Basisklasse `struct Exp{}`
- Abgeleitete Klasse `struct BinExp: public Exp{}`
- Abstrakte Klasse `struct Exp{virtual int size() const = 0...}`
- Polymorphie `virtual double eval()`

Ⓑ

- Ausdrucksknoten und Erweiterungen

Ende

Ende der Vorlesung.